ON DEPENDENT DATA FETCHING IN CLOUD ENVIRONMENT

A Thesis

by

JIACHENG GU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jianer Chen
Committee Members,	Sergiy Butenko
	Anxiao (Andrew) Jiang
Head of Department,	Dilma Da Silva

December 2015

Major Subject: Computer Science

ABSTRACT

This work was motivated by the study of the file fetching process in a cloud system, in particular by the recent progress in the model of transparent computing. A transparent computing system may have many clients, each requesting a significant set of files from the server, including user data and many commonly used softwares (operating systems and apps). These files may have inherent dependence relations so should be received by the clients in a specific topological order. On the other hand, since many of these files are commonly used softwares, many clients may request copies of the same files. This proposes an interesting problem on the server side of how this kind of requests should be handled efficiently to improve the performance of the system. In particular, we are interested in the processes that significantly reduce the disk IO operations in the server, which are in general very time-consuming. We propose a formal model for this problem and study its validity and correctness. Heuristic algorithms for the problem are proposed and studied. Simulation results are presented to compare the proposed heuristics and algorithms based on known techniques in scheduling literature. 7% - 20% of the total disk IO can be reduced via the optimizations proposed in this work.

DEDICATION

To my parents, who always believe in me.

To my cat, who never cares.

To my girlfriend, who consistently supports me, especially when I am frustrated.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Dr. Jianer Chen for his continuous support throughout my studies at Texas A&M University. His guidance, patience and insights helped me in all the time of writing this thesis. I could not have imagined having a better advisor for my master study.

Meanwhile, I would like to thank my committee members, Dr. Sergiy Butenko and Dr. Anxiao Jiang, for their valuable suggestions and encouragement. My thanks also goes to Dr. Kehua Guo, for all the time working together at the office and for all the discussions we had about or beyond the project.

Lastly, I want to thank my family for always believing in me.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

As the emerging of Cloud Computing, the job scheduling has become a critical operation for the performance and utilization of clusters. A typical cluster, no matter it is applied with distributed file system like Google File System (GFS) [6] or cloud computing platform like Microsoft Azure, is faced with massive amount of task requests continuously. Meanwhile, these task requests tend to have a huge diversity concerning the computing power and hardware resources that need to be utilized.

Take Transparent Computing (TC) [22] as an example to illustrate the challenges in large-scale cluster scheduling. TC is an emerging service-oriented computing paradigm that aims at serving users with heterogenous OSes and applications on terminal devices. Users are not required to pre-install any OSes or applications on their devices, while all codes and files will be dynamically loaded from the clusters upon users' needs. The amount of requests from a single user can be considerable, especially when they log in a device for the first time. Meanwhile, these requests have huge diversity in completion time. Some may request for configuration files with size below KB, while some may request for OS images with size beyond GB. Therefore, the scheduling's efficiency greatly determines whether the cluster can fit such scalability in the amount and volume of task requests.

The increasing quantity and complexity of requests raise more potential for the optimization of scheduling algorithms. There are two major facts that motivate this work.

- The dependency between subjobs freqsubjobsuently exist. In a system with service-oriented architecture (like TC example above), users deliver their service requests to the cluster and these requests can be further decomposed into

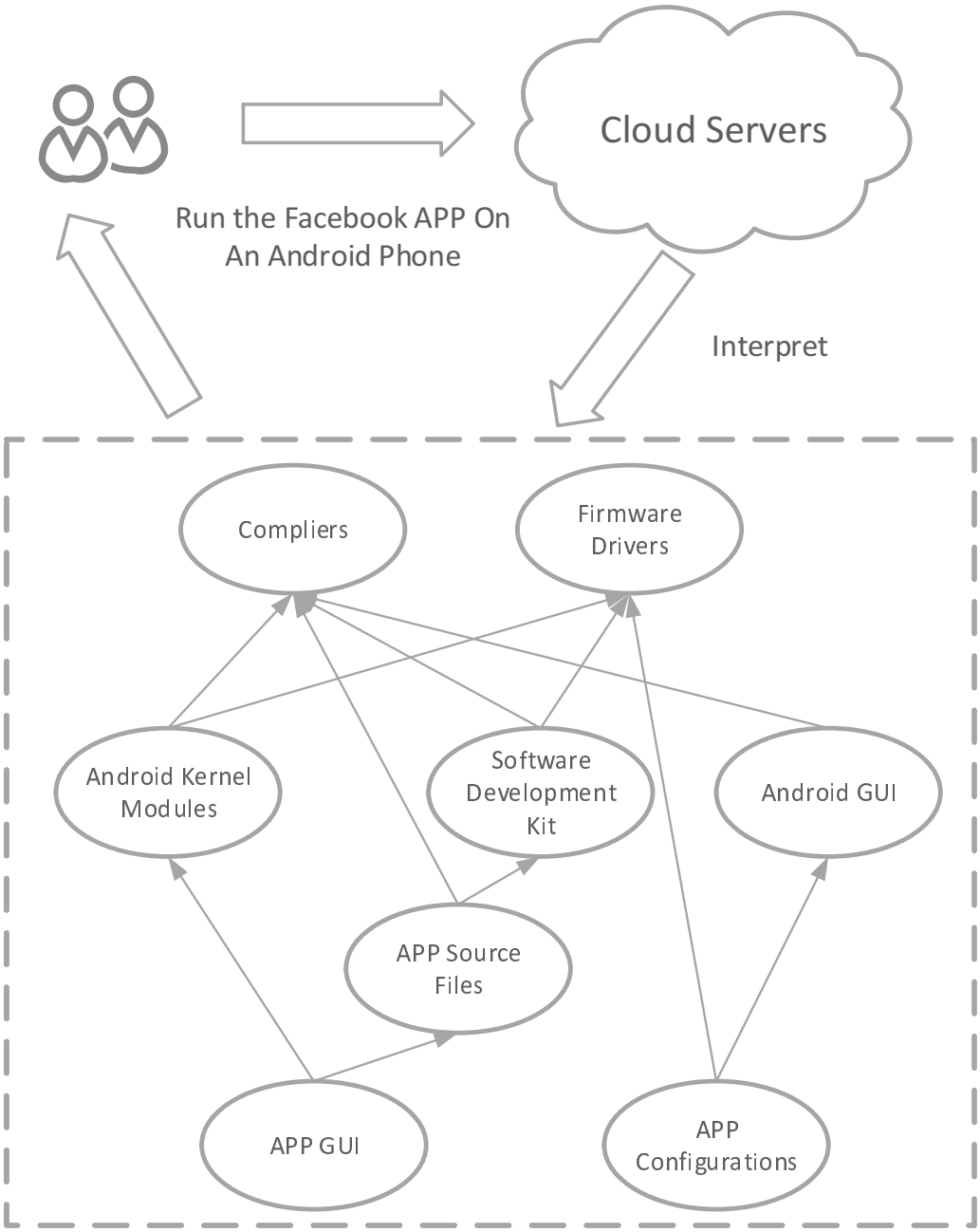

Figure 1.1: Transparent Computing Request Pattern

a series of subjobs with inter-dependencies. Similar case holds for database systems with SQL-based query language like SCOPE [2]. A SCOPE execution can be parsed into several subjobs which are presented in a directed acyclic graph (DAG). Figure 1.1 shows how the complicated tasks are presented in the cluster.

- Some requests from the clients are identical. Microsoft proposed their scheduling framework Apollo [1] in 2014. Apollo is designed to handle clients' requests at a expecting rate more than 50k requests/sec. Although the user behaviors may pull various requests, it is a strong argument that there exist requests for identical data among such a considerable number of base. Such identical requests appear more frequent in the TC system or clusters working as a application store. Users tend to frequently request OS images and applications. Some files will become 'hot spots' due to the newly released updates for popular applications.

In this work, we fully consider the challenges for the scheduling process in a cluster and aim at optimizing it. The major target for the optimization is to explore identical tasks and schedule them in a manner that their overall cost, like disk IO or computing time, will be reduced. Meanwhile, the inter-dependency between tasks and the performance on the client's side are also highly valued during the optimization. Following are the contributions of this work.

- Abstracts a scheduling model based on Transparent Computing architecture. The model serves file requests from multi-clients and schedule these requests to servers in the cluster.

- Focuses on the file fetching process in awareness of the cost saving which is brought by identical file requests from various clients.

- Heuristic algorithms are proposed to optimize the file fetching process.
- The heuristic algorithms are evaluated under the scheduling model, in comparison with traditional scheduling methods.

The rest of the thesis is organized as follows. Section 2 includes the related works for this research. Section 3 presents the scheduling model used for the file fetching process and defines two scheduling problems — Single-Server Valid Sequence Generation (VSG) problem and Multi-Server Job Partition (JP) problem. Then section 4 proposes heuristic algorithm for these two problems. In section 5, the heuristic algorithms are evaluated in reference to other used solutions for file fetching process. Section 6 will cover some future topics concerning the model proposed in the work.

2. RELATED WORK

The studies on job scheduling are always the core of performance optimization in computer system. There exist some classic problems that can be adapted to cloud scheduling and they significantly inspire this work.

1. Two and three stages production scheduling [10] is proposed by Dr. S. M. Johnson. The work proposed the decision rule that leads to an optimal scheduling solution for the two-stage case. A restricted case of three stage problem was also solved [10].

2. Flow shop sequencing problem [14] [16] extended the scheduling problem to a more general $n \times m$ flow shop problem.Considering the heuristic methods that have been proposed for this problem, the major objectives are usually minimizing the overall makespan or minimizing the idle time of underlying machines.

3. List scheduling [20] is another scheduling topic that are frequently used in instruction scheduling on multiprocessors [3] and scheduling upon heterogeneous systems [17]. List scheduling problem adapts direct acyclic graphs (DAG) to represent inter-dependencies of jobs, keeps a ready queue for nodes that are ready to be processed and assigns priorities to nodes in DAG to help deciding the processing order. These features siginificantly inspires this work.

In this work, the optimization of file fetching process is defined as a sequencing problem upon multiple Directed Acyclic Graphs (DAGs).

Moreover, this work is motivated by the real need in Transparent Computing (TC) Project. TC follows a Service-Oriented Architecture (SOA) [23]. Users request

all services from the cloud without storing any OSes or applications locally. Such services are secured by pervasive network across heterogeneous software and hardware platforms [24]. Figure 2.1 shows a basic service architecture of TC.

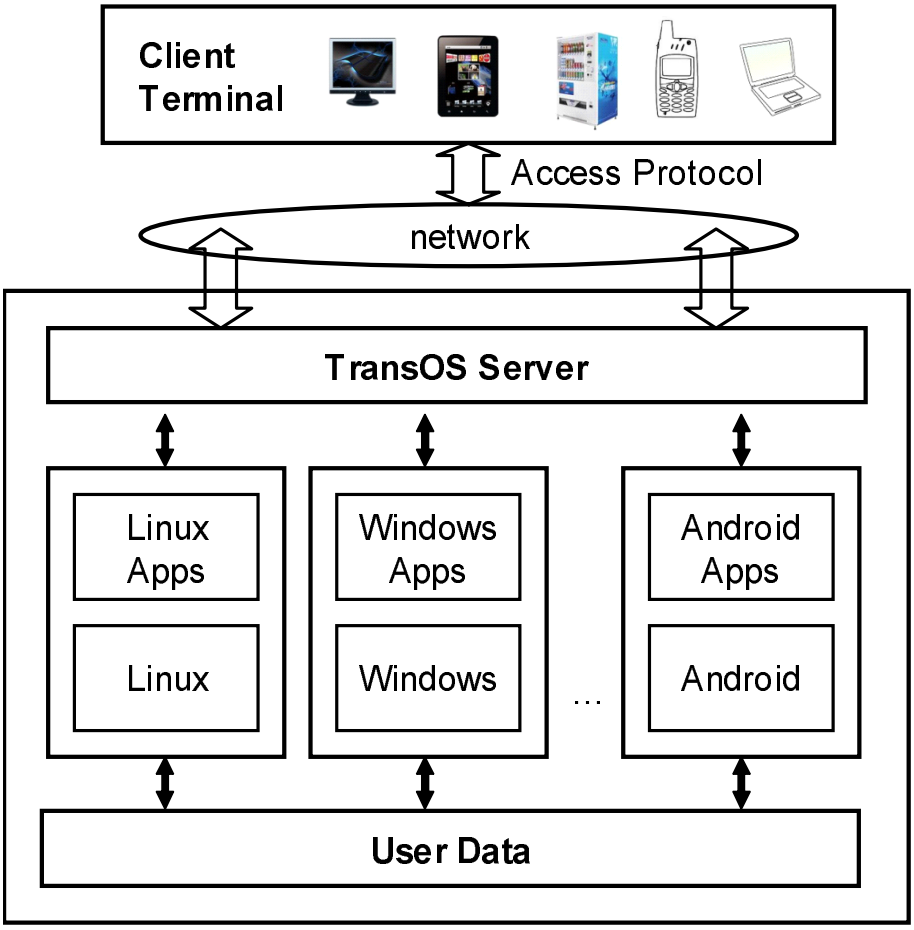

Figure 2.1: Transparent Computing Architecture

As shown in figure 2.1, there are two key features, via which users can get pervasive access to TC service. One is TransOS (Transparent Operating System) server and the other is pervasive network. TransOS is responsible for access control, job scheduling and data management. Client terminals are light-weighted devices which

only needs to locally store Basic Input and Output System (BIOS) and a small fraction of booting protocols for the devices.

Concerning the design and implementation of TC infrastructure, the study [11] points out critical issues in scheduling problems. Therefore, the research will validate and evaluate the scheduling algorithms in an infrastructure that is abstracted with reference to Transparent Computing [22], Apollo from Microsoft [1] and GFS from Google [6]. The data fetching process is the major perspective and the research goal is to minimize the file fetching cost as much as possible via taking benefits from those identical requests from various clients.

Figure 2.2 represents the scheduling model used for this work. More details can be found in Scheduling Model section. Centralized metadata severs (or *Job Manager*) is a common solution nowadays for data center. A wide range of researches has been proposed upon the design of *Job Manager* or *Metadata Server*. *Resource Monitor*, as part of *Job Manager*, involves topics like dynamic resource provisioning and shared resource pooling [13] [7]. Technical solutions to design an efficient *Resource Monitor* includes hierarchy listen/announce protocol [12] or peer-to-peer self-organizational protocol [18].

To optimize the file fetching process, finding the bottleneck is the first task to start with. As proposed in [21], performance barriers emerge along with salient features of cloud environment. One of them is the high-volume I/O due to server consolidation and scalability in expansion. Both disk I/O and network bandwidth are critical to the file fetching process in this work. Moreover, compared to network bandwidth, the Hard Disk Drive (HDD) has a slower growth in its bandwidth (close to the physical constraint). Solid State Drive (SSD) does bring a huge improvement in local storage performance. However, its application is limited by its high cost and cannot replace HDD as the major storage in the current data centers. Therefore,

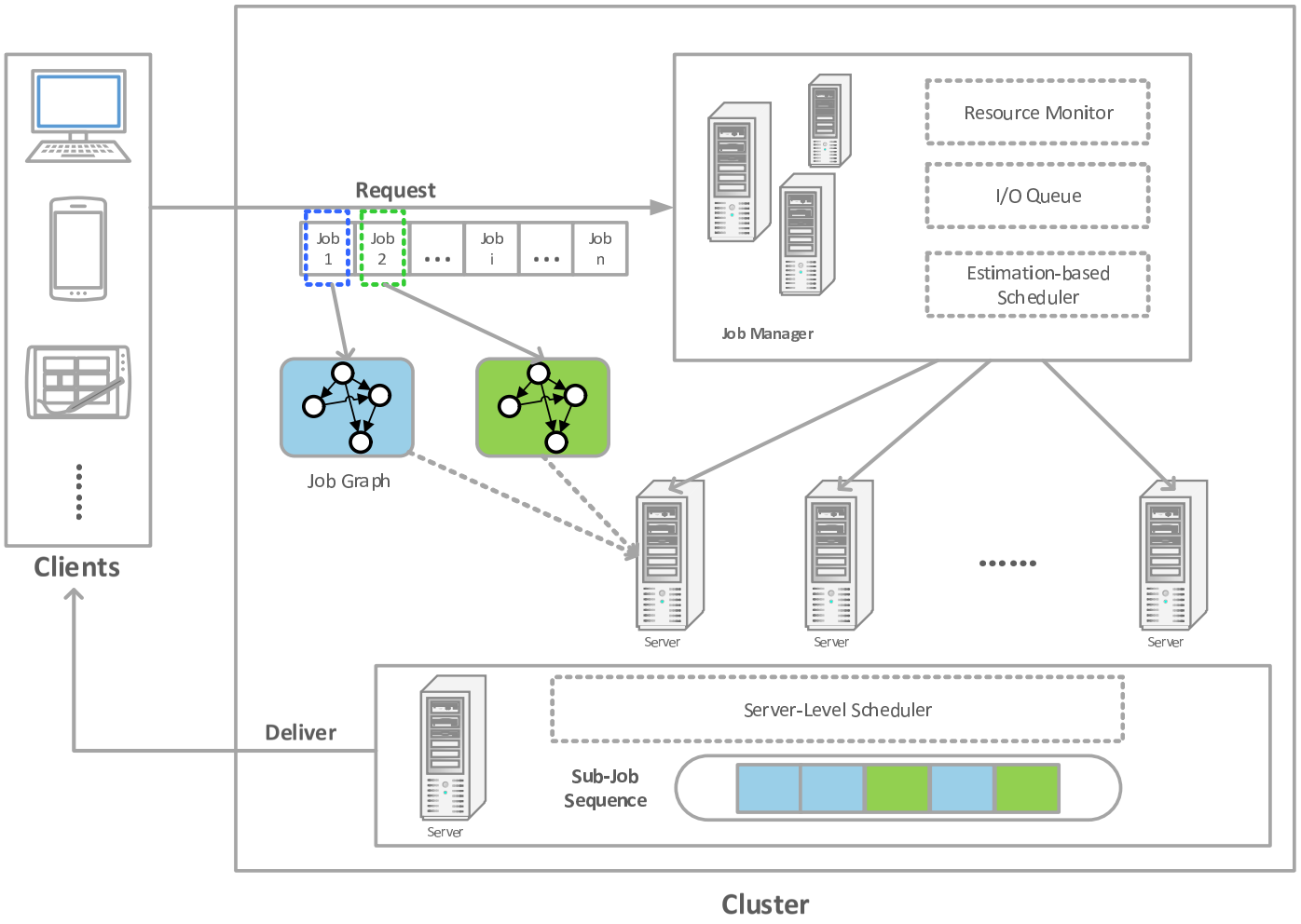

Figure 2.2: Scheduling Model for File Fetching Process.

in this work, the major barrier to be optimized will be the disk I/O. Although the mathematical model is formatted in a more general way, the original motivation is to save the disk I/O.

Therefore, job scheduling is considered as the breakpoint for such I/O bottleneck. There are diverse researches on estimation based scheduler. Such scheduler assigns jobs to underlying servers based workload estimation. The *Job Manager* dynamically collects information of cloud resources (latency, bandwidth, queue status, etc) via *Resource Monitor*. A classic algorithm used to balance the workload is stable matching algorithm [5]. On the server side, most infrastructures use regular topological sorting or either execute all the tasks in a FIFO manner.

3. SCHEDULING MODEL AND PROBLEM DESCRIPTION

3.1 Scheduling Model

The services based on cloud computing are diversified. The definition and taxonomy of cloud computing vary rapidly as new services are emerging all the time. However, most of these services are faced with the challenges mentioned in the last section: the amount of users and the cost of jobs scale considerably for each server in the cluster. In this section, a scheduling model will be presented and a formal definition upon the file fetching problem in the cloud environment will be given.

Figure 2.2 presents the model that will be used along this work. The model is abstracted from common architectures of cloud-based services [15] [22] [4]. First of all, the model in Figure 2.2 follows a basic Service Oriented Architecture (SOA), where client's requests are addressed in the view of 'services' and it is the cluster's responsibility to interpret service requests into concrete jobs. The types of clients are not limited here; it can be laptops, smart phones, tablets or any devices capable to get access to the system. Meanwhile, the network connections between the clients and the cluster are not limited to any specific types of access as well. Yet the performance of the network IO does constrain the optimization, which will be discussed in the later sections.

On the cluster side, the first step upon receiving a service request is to interpret it into a corresponding job graph G_i, which includes all the subjobs needed to accomplish the request. Meanwhile, G_i will be delivered to *Job Manager* (JM), which has three major components related to the scheduling process — *Resource Monitor*, *I/O Queue* and *Estimation-Based Scheduler*. The first one is *Resource Monitor*. Its job lies in keeping track of the performance and availability of underlying servers.

Meanwhile, the *Resource Monitor* will be responsible to collect heartbeat messages within the cluster. These messages can bring periodical feedbacks concerning the workloads and failure/recovery status. The I/O queue stores and manages all the job requests and its corresponding information like deadline, request source and most importantly the job graphs (formatted as DAGs) . Then the estimated-based scheduler will schedule the receiving jobs based on their cost estimations and workload feedback from *Resource Monitor*.

Upon receiving job assignments from the *Job Manager*, the server will schedule all the subjobs from received job graphs into its local sequence. This is the major process this work will focus on. When the amount of jobs is huge and identical subjobs widely exist, the scheduling on a single server has the potential to be optimized.

In summary, when a job is requested to the cloud, it will go through two scheduling stages.

1. **Partition.** Happens at *Job Manager*. JM will interpret all the requests it received into DAGs. Then distribute these DAGs to underlying servers based on criteria like job cost, server's workload and latency etc.

2. **Sequencing.** Happens at each single server. The server will schedule all the DAGs it received from JM into a local sequence. Then the server will execute all the subjobs following the sequential order.

Here are some essential concepts related to the scheduling optimization.

- *Job Graph.* Each Job from clients is interpreted as a Directed Acyclic Graph (DAG). Each vertex in the DAG represents a subjob and each directed edge indicates the dependency relationship between two subjobs.

- *Subjob Sequence*. The sequence includes all the subjobs (vertices) from all the job graphs (DAG) received on one server and can be executed in serial order.
- *Weight*. Each subjob (vertex) will be assigned with a *Weight* value which will help the scheduler determine which subjob should be executed next.

For a more general description, a job can be a service request from a client like "Update and Run Facebook Application on an Andriod Phone". Meanwhile, this job will be interpreted as a DAG G_i, where each vertex represents a subjob. Each subjob includes fetching and delivering one specific file for the client.

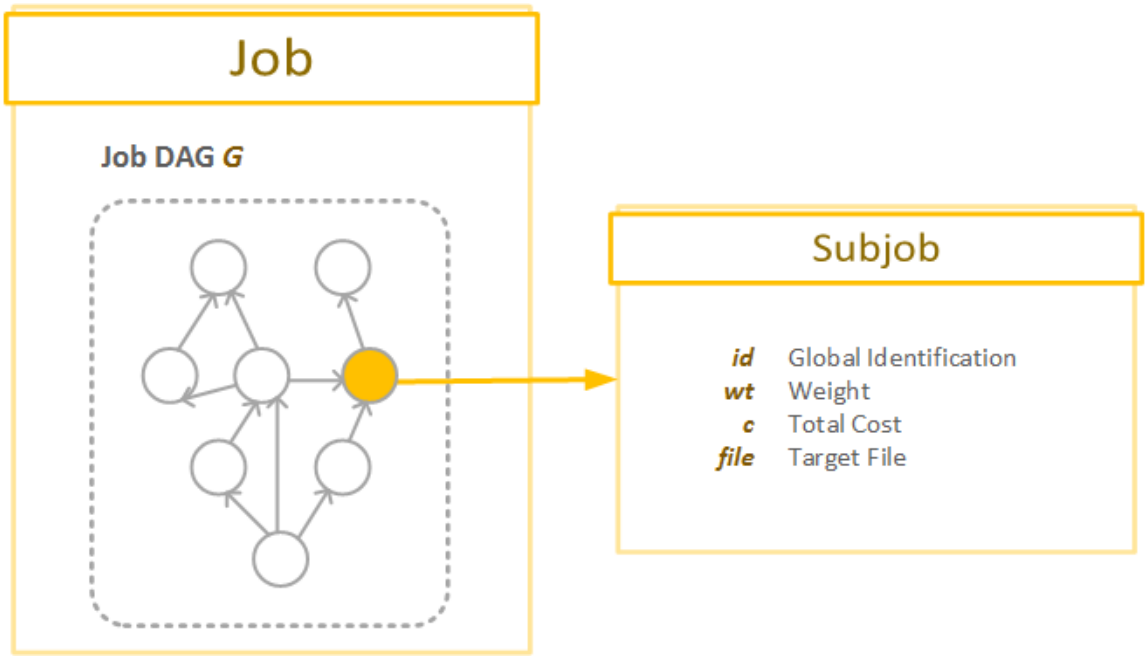

Figure 3.1: Job and Subjob

3.2 Single-Server Scheduling

Figure 3.1 illustrates the basic information concerning job and subjob in the model. Each job is requested by one source client for a service request.

During the partition stage, the JM will interpret jobs into DAGs and deliver these DAGs to the underlying m machines via estimation-based scheduler, where the server set is $\Omega = \{M_1, M_2, ..., M_m\}$. A single server is assumed to have n job DAGs needed to be scheduled and executed. We define the graph set Γ on a single server as

$$\Gamma = \{G_1, G_2, ..., G_n\} \tag{3.1}$$

A single job (interpreted as G_i) consists of k subjobs. Each subjob had a unique identification in the set Γ and is associated with a target file. Note that two subjobs, which can be either in the same job or in difference jobs in Γ, may be associated with the same target file. Each subjob will have a cost c calculated according to its context in a scheduled sequence S, which represents the cost of fetching the target file from disk into memory buffer. (See the following discussion for more details.)

The local scheduler of each server will generate a subjob sequence S for file fetching process. The validity of the sequence is defined in definition 3.1

Definition 3.1. ***Valid Sequence:*** *Given a set of job DAGs $\Gamma = \{G_1, G_2, ..., G_n\}$, let V_Γ be the collection of vertices and E_Γ be the collection of edges in all DAGs in set Γ. A valid subjob sequence $S = [s_1, s_2, ..., s_N]$ must satisfy:*

1. *Elements in the sequence S and subjobs in V_Γ follow a bijective map function $\pi : S \to V_\Gamma$. Each subjob vertex u in V_Γ pairs with exactly one element in the sequence S.*

2. *For any edge $e(u, v) \in E_\Gamma$, where $u = \pi(s_i)$ and $v = \pi(s_j)$, we must have $i > j$.*

Based on definition 3.1, the following lemmas can be further derived

Corollary 3.2. *For any path that can be found in a DAG in Γ starting at vertex $u = \pi(s_i)$ and ending at target vertex $v = \pi(s_j)$, we must have $i > j$.*

Proof. The path in any DAG is composed of a series of vertices and directed edges. Definition 3.1 (2) holds for all edges and its transitive feature will be contradicted if $i > j$ holds for any path in Γ. □

Each server has a buffer B in main memory with certain pre-given size. Files will be fetched from disk drive to buffer B and wait for further transmissions. When a subjob s_i is being executed during the file fetching process, its cost will be reduced in the following scenario.

- If the target file of s_i has been already loaded and still held in the buffer due to foregoing subjobs, there is no need to fetch it again from disk drive and it can be directly transmitted to the source client.

Therefore, when a subjob s_i is being executed, the context of buffer B is essential for reducing cost. We introduce the concept of *segment* to help illustrating the file fetching process. To begin with, here are some notations concerning the file fetching process and will be further used along this work.

- $f(s_i)$ refers to the target file associated with the subjob s_i.
- $f(s_i).size$ indicates the size of memory space needed to accomplish s_i.
- $s_i.source$ refers to the DAG that contains s_i.
- $B.size$ denotes the total memory size reserved for buffer B.

Definition 3.3. ***Segment*** *Given a valid sequence $S = [s_1, s_2, ..., s_N]$, a segment $S_{i,j}$ is a subsequence of S, represented as $[s_i, s_{i+1}, ..., s_j]$ where $i \leq j$.*

Here are some features concerning a segment $S_{i,j}$.

- Let set $F_{i,j}$ be the file set that includes all the target files required for subjobs in $S_{i,j}$, i.e. $F_{i,j} = \bigcup_{k=i}^{j} \{f(s_k)\}$.
- We define $S_{i,j}.size$ as the total size of all file objects in $F_{i,j}$, i.e $S_{i,j}.size = \sum_{f \in F_{i,j}} f.size$. $S_{i,j}.size$ denotes the amount of memory size needed to accomplish all subjobs in the segment $S_{i,j}$.

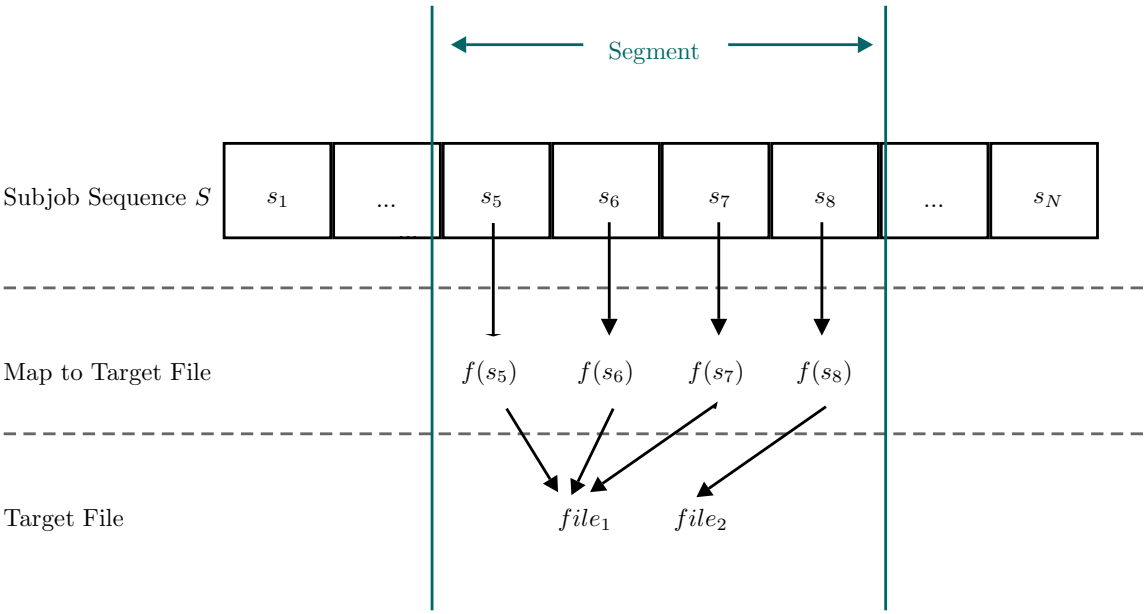

Figure 3.2: Segment Example. $S_{5,8} = [s_5, s_6, s_7, s_8]$, where $F_{5,8} = \{file_1, file_2\}$ and $S_{5,8}.size = file_1.size + file_2.size$.

Figure 3.2 presents an example of the concept *segment*. With the definition of *segment*, we can now move on to discuss the cost of each subjob in S. When the buffer B is bounded with a certain size, the amount of files existing in B will become restricted. A bounded segment is applied to help calculating the cost of a subjob s_j.

Definition 3.4. ***Subjob Cost*** *Given a valid sequence $S = [s_1, s_2, ..., s_N]$, when calculating the cost for s_j, let $S_{i,j}$ be the segment with the smallest possible i such*

that $S_{i,j}.size \leq B.size$. *Then the cost of the subjob* s_j *will be calculated as*

$$s_j.cost = \begin{cases} 0 & \exists k \ (i \leq k < j), \ where \ f(s_k) = f(s_j) \\ f(s_j).size & others \end{cases}$$

The total cost for the sequence will be

$$Cost_S = \sum_{i=1}^{N} s_i.cost \tag{3.2}$$

It need to be noticed that we assume the sizes of any files requested will not have a size larger than the buffer size. This assumption will be held along this work.

The generation of the valid sequence will be the major focus of this work. To optimize the valid sequence generation, the major target will be minimizing the overall cost of all n *subjobs* in a single server via taking benefits from same files requested from different clients. In other words, the sequence generated for a single server should maximize the possibility that target files requested can be found in the memory buffer without fetching them from disk drives. The problem can be formally defined as following.

Definition 3.5. ***Single-Server Valid Sequence Generation (VSG) Problem:*** *Given a job DAG set* $\Gamma = \{G_1, G_2, ..., G_n\}$ *for a single server and a memory buffer* B *with a fixed size, generate a valid sequence* S *including all subjobs and satisfy all the dependencies in set* Γ *while the overall cost of the sequence is minimized.*

The difficulties of the problem can be clarified in two aspects. Firstly, the amount of valid sequences for a single graph G_i can be diverse. The order of tasks significantly depends on the implementation of topological sorting algorithm used. Moreover, in this work, generating a valid sequence from a set of graphs Γ brings larger set of

possible solutions. In the following sections, we use a heuristic way to solve the problem and evaluate it upon the scheduling model defined.

Here is a simple example for the file fetching process to illustrate how the sequence generation will affect the overall cost. Figure 3.3 includes two job requests in a single server.

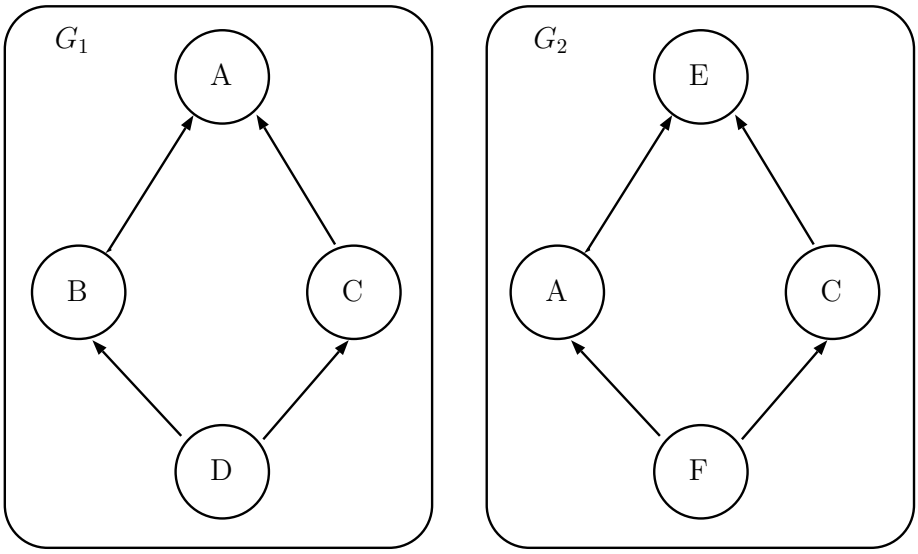

Figure 3.3: File Fetching Cost Example

Table 3.1: Target File Size

Target File	A	B	C	D	E	F
Size(MB)	100	240	170	20	10	35

We denote each subjob here with its file name and the DAG it belongs to. A_1 denotes a subjob that has target file A and belongs to job DAG G_1. With DAG set $\Gamma = \{G_1, G_2\}$, both of the following sequence will be valid according to definition

3.1.

$$S_1 = [A^{(1)}, C^{(1)}, B^{(1)}, D^{(1)}, E^{(2)}, A^{(2)}, C^{(2)}, F^{(2)}]$$
$$S_2 = [E^{(2)}, A^{(1)}, C^{(1)}, A^{(2)}, C^{(2)}, B^{(1)}, D^{(1)}, F^{(2)}]$$

Assume a buffer B with size 300 MB is reserved in the main memory. The cost for each sequence is calculated based on definition 3.4 and the results are included in table 3.2

Table 3.2: Cost for Two Jobs with Difference Sequence

S_1	$A^{(1)}$	$C^{(1)}$	$B^{(1)}$	$D^{(1)}$	$E^{(2)}$	$A^{(2)}$	$C^{(2)}$	$F^{(2)}$	Total Cost
Size (MB)	100	170	240	20	10	100	170	35	845
Reduced Cost?	No	No	No	No	No	No	No	No	

S_2	$E^{(2)}$	$A^{(1)}$	$C^{(1)}$	$A^{(2)}$	$C^{(2)}$	$B^{(1)}$	$D^{(1)}$	$F^{(2)}$	Total Cost
Size (MB)	10	100	170	100	170	240	20	35	575
Reduced Cost?	No	No	No	Yes	Yes	No	No	No	

There are two subjobs that achieve reduced costs with S_2. Let us have detailed views on each of them.

- For the subjob $A^{(2)}$ in S_2, we can have a segment $S_{2,4} = \{A^{(1)}, C^{(1)}, A^{(2)}\}$ and the corresponding file set $F_{2,4} = \{A, C\}$. Since subjob $A^{(2)}$ has a target file A, where $A^{(1)}$ has the same target file and it is included in $S_{2,4}$, the cost for $A^{(2)}$ can be reduced.

- Similar case holds for $C^{(2)}$. A segment $S_{2,5} = \{A^{(1)}, C^{(1)}, A^{(2)}, C^{(2)}\}$ can be derived and its file set $F_{2,5} = \{A, C\}$. The cost of $C^{(2)}$ will be reduced due to the existing $C^{(1)}$ in $S_{2,5}$.

3.3 Multi-Server Scheduling

Beyond the sequence generation on a single server, another important issue for the cloud to deal with file fetching process is distributing all the DAGs to underlying servers. As discussed in the Scheduling Model section, *Job Manager* will be responsible for interpreting all the jobs received into DAGs and assigning them to the servers. This process can be turned into a partition problem, where the core objective will still be reducing the disk IO cost of all jobs scheduled and balancing the workloads among all servers.

Consider we have a set of job DAGs $\Gamma = \{G_1, G_2, ..., G_n\}$ and a set of underlying servers $\Omega = \{M_1, M_2, ..., M_m\}$. We need to partition the set Γ into m subsets that will be represented as $\Gamma_i = \{G_1^{(i)}, G_2^{(i)}, ..., G_{n_i}^{(i)}\}$ $i = 1, 2, ..., m$, where $\Gamma = \bigcup_{i=1}^{m} \Gamma_i$ and $n = \sum_{i=1}^{m} n_i$. Then for each server M_i, it will generate a valid sequence from the set Γ_i as discussed in the previous sections and $Cost(\Gamma_i, M_i)$ will be used to indicate the total disk IO cost of executing all jobs in Γ_i upon the server M_i. Note that for each $Cost(\Gamma_i, M_i)$, the actual value of it really depends on the algorithm each server uses for the sequence generation and the buffer size reserved for file fetching. In this work, we assume all the underlying servers are identical for the simplicity, where all server will have the same capability in disk IO, memory buffer and the same algorithm applied for sequence generation. Therefore, the cost of each job will be proportional to its file size. Then the problem can be formally defined as

Definition 3.6. ***Multi-Server Job Partition (JP) Problem*** *Given a job DAG set* $\Gamma = \{G_1, G_2, ..., G_n\}$ *and a server set* $\Omega = \{M_1, M_2, ..., M_m\}$*, partition* Γ *into* m *subsets* $\Gamma_1, \Gamma_2, ..., \Gamma_m$*, where the largest disk IO cost among all* $Cost(\Gamma_i, M_i)$ *is minimized.*

4. SOLUTION FOR THE VSG PROBLEM

In this section, a preliminary section will cover the specifics of the VSG problem like validity check and cost calculation. Then a short survey is presented to show the possible solutions for the VSG problem. Lastly, we propose a heuristic algorithm that aims at reducing overall cost of a subjob sequence.

4.1 Preliminary

To begin with, since the algorithms discussed here are mostly related to DAG and queue operations, some frequently used notations will be introduced first.

- $v.indegree$ indicates the number of nodes having edge (u, v) that points to vertex v. If $v.indegree = 0$, the vertex v represents a subjob that no other subjobs depend on it.

- $v.outdegree$ indicates the number of nodes u which node v has outgoing edges (v, u) point to. If $v.outdegree = 0$, the vertex v is considered as a leaf vertex and does not depend on other subjobs.

- Ready Queue Q is a global array that includes all the vertices ready to be put into the sequence. All the elements v in Q satisfy $v.outdegree = 0$.

The subjob sequence S must satisfy all the dependency relations that exist in set Γ. Algorithm 1 is proposed to verify the validity of the sequence upon the set Γ. The algorithm traverses the whole sequence S from s_1 to s_N and each subjob should have all its precedent subjobs being executed prior to it.

Note that the function $\pi : S \rightarrow V_\Gamma$ maps each subjob s_i in S to a vertex in some DAG in Γ. According to definition 3.1(2), a subjob s_i will be ready to be

Algorithm 1 Validity Verification

Input: Subjob Sequence $S = [s_1, s_2, ..., s_N]$; Job Set Γ; Map Function $\pi : S \to \Gamma$
Output: Whether the sequence S is valid or not;

```
for i = 1 to N do
  if π(s_i).outdegree! = 0 then
    return false; {s_i depends on tasks that will be executed after it in sequence}
  end if
  Let G_k ∈ Γ be the source DAG that contains vertex π(s_i).
  for each directed edge (u, π(s_i)) in G_k do
    u.outdegree − −;
  end for
end for
return true;
```

executed only when all the subjobs it depends on have been executed. Algorithm 1 uses $\pi(s_i).outdegree$ to trace the number of subjobs that s_i depends on and still remains uncompleted.

Upon verifying the validity of a sequence, the cost of sequence has several features which will help the following discussion. To begin with, for two consecutive subjobs associated with a same target file in the sequence, their overall cost will be reducible.

Lemma 4.1. *Given a sequence $S = [s_1, s_2, ..., s_j]$ and a subjob s_{j+1}, let $F_{i,j}$ be the file set for the segment $S_{i,j}$ with the smallest i holding $S_{i,j}.size \leq B.size$. Then if $f(s_{j+1}) \in F_{i,j}$, we have*

1. *The sequence $S' = [s_1, s_2, ..., s_j, s_{j+1}]$ will have the same cost as S.*

2. *The file set $F_{i,j+1}$ for the segment $S_{i,j+1}$ is the same as $F_{i,j}$ for $S_{i,j}$.*

Proof. With two segments $S_{i,j} = [s_1, s_2, ..., s_j]$ and $S_{i,j+1} = [s_1, s_2, ..., s_j, s_{j+1}]$, their corresponding file sets will be $F_{i,j}$ and $F_{i,j+1}$, where $F_{i,j+1} = F_{i,j} \cup \{f(s_{j+1})\}$. Since $f(s_{i+1}) \in F_{i,j}$, we will have $F_{i,j} = F_{i,j+1}$. Moreover, according to definition 3.4,

$s_{j+1}.cost = 0$ if there is a subjob s_k in segment $S_{i,j}$, where $f(s_k) = f(s_{j+1})$. Therefore, since $Cost_{S'} = Cost_S + s_{j+1}.cost$, we must have S and S' with the same cost. □

More generally speaking, if $f(s_{j+1}) \in F_{i,j}$, appending any subjob s_{j+1} to the sequence S will neither add extra cost to the sequence nor change the file set $F_{i,j}$ used to calculate the cost. Moreover, given a valid sequence S and a memory buffer B, the sequence generated will have upper and lower bounds as shown in Theorem 4.2.

Theorem 4.2. *Given a valid sequence $S = [s_1, s_2, ..., s_N]$ and a fixed size buffer B, the overall execution cost will follow the bounds as*

$$\sum_{f \in F_{1,N}} f.size \leq Cost_S \leq \sum_{i=1}^{N} f(s_i).size \tag{4.1}$$

where $F_{1,N}$ is the file set that includes all the target files needed in S, i.e. $F_{1,N} = \bigcup_{i=1}^{N} \{f(s_i)\}$.

Proof. For the lower bound, it can be derived from the case where an infinite large buffer is applied. With an infinite large buffer, all the files will be fully fetched only once and all the later subjobs associated with same files can take advantage of existing target files. The lower bound is derived as

$$Cost_S \geq \sum_{f \in F_{1,N}} f.size$$

For the upper bound, we can assume a buffer size only be able to hold one target file each time. Each s_i need to be accomplished via loading the target file into buffer. In such case, each task will have a full time cost without taking advantage of same target files between different subjobs. Then the upper bound can be represented as

following.

$$Cost_S \leq \sum_{i=1}^{N} f(s_i).size$$

□

When calculate the cost for each subjob s_j, it is essential to construct a segment $S_{i,j}$ with the smallest i satisfying $S_{i,j}.size < B.size$. Algorithm 2 is proposed to construct such a segment $S_{i,j}$ to help calculate the cost for the subjob s_j.

Algorithm 2 Longest Segment (LONG-SEG)

Input: Sequence $S = [s_1, s_2, ..., s_N]$; Memory Buffer B; Subjob s_j
Output: Segment $S_{i,j} = [s_i, s_{i+1}, ..., s_j]$;

```
F ← empty set for target files;
size = 0;
for i = j to 1 do
  if f(s_i) ∉ F then
    Add f(s_i) to F;
    size = size + f(s_i).size;
  end if
  if size + f(s_{i-1}) > B.size and f(s_{i-1}) ∉ F then
    break;
  end if
end for

return segment S_{i,j} = [s_i, s_{i+1}, ..., s_j];
```

Next, we will start to discuss the construction of a sequence with a job set Γ given. Suppose now we have a subjob s_{j+1} selected from a ready queue Q and want to append it to partial sequence $S = [s_1, s_2, ..., s_j]$. Algorithm 3 illustrates how to append the selected subjob to the sequence and update the ready queue Q. Note

that the vertex set V_Γ and the edge set E_Γ include all the vertices and edges in set Γ. Meanwhile, the ready queue Q is a global array which includes all the subjobs satisfying $outdegree = 0$.

Algorithm 3 Update Sequence and Ready Queue (UPDATE)

Input: Subjob s_{j+1}; DAG Set Γ; Sequence $S = [s_1, s_2, ..., s_j]$;
Output: Sequence $S' = [s_1, s_2, ..., s_j, s_{j+1}]$;
  for each edge $e(v, \pi(s_{j+1})) \in E_\Gamma$ **do**
    Remove e from E_Γ;
    $v.outdegree - -$;
    if $v.outdegree = 0$ **then**
      Add v to the ready queue Q;
    end if
  end for
  $S' = S + s_{j+1}$; {where $\pi(s_{j+1}) = u$}

4.2 A Heuristic Algorithm for VSG Problem

In this section, a specific heuristic algorithm for VSG problem is proposed aiming at taking advantage of those subjobs associated with same target files. The core idea is always choosing the subjob which can save the most cost for the current step and adding it to the sequence. Therefore, the algorithm will follow a greedy strategy. A weight value w is assigned to each subjob s_i to indicates how many subjobs are depending on s_i. A typical way to define the weight value of a subjob s_i with $\pi(s_i) = u$ will be

$$wt(s_i) = u.indegree$$

Here, the weight $wt(s_i)$ of each subjob will be the number of subjobs directly depend on it. There are more ways to define the weight values which will be further discussed in the following content. The core idea will be the same — higher the weight value one subjob s_i has, more subjobs are currently depending on s_i.

Now, we can move on to discuss the heuristic solutions for the VSG problem. Remind that the sequence generation process can be interpreted as picking subjobs from Γ and continuously append them to the sequence. To ensure the validity of the sequence generated, the following heuristic algorithm will only pick subjobs from the *Ready Queue* Q, where all its elements satisfy $outdegree = 0$. Here is the strategy applied to the heuristic solution to decide which subjobs should be picked next.

- For each subjob s' in the ready queue Q, we define *direct reduction (DR)* in cost with sequence $S = [s_1, s_2, ..., s_j]$ given.

$$DR\ (S, s') = \begin{cases} f(s').size & f(s') \in F_{i,j} \\ 0 & \text{Others} \end{cases}$$

  where $F_{i,j}$ is the file set for the longest segment $S_{i,j}$ achieved from algorithm 2, i.e. $S_{i,j}$= *LONG-SEG*(S, B, s_j) .

  According to lemme 4.1, if any subjobs in Q is assigned with a positive *direct reduction*, adding it as s_{j+1} to the sequence S will not increase the overall cost of the sequence.

- Meanwhile, we define *indirect reduction (IR)* for each subjob s' in the ready queue Q. The *indirect reduction* of subjob s' indicates the amount of *direct reduction* that will be created via adding s' to the sequence. Thus, for each s' in the ready queue, let $S' = [s_1, s_2, .., s_j, s_{j+1}]$, where $s_{j+1} = s'$, and let

$Q' = Q \backslash \{s'\}$, we have

$$IR(S, s') = \sum_{s'' \in Q'} DR(S', s'')$$

Without the presence of positive *direct reductions* in Q, the algorithm will always choose the subjob s' with the largest *indirect reduction*, puts it to the sequence and starts over to look for subjobs with *direction reduction* in Q.

- If neither of positive *direct* and *indirect* reduction can be found, the algorithm picks one subjob s' from the ready queue Q with the largest weight. We define *estimated reduction (ER)* for each subjob s' in Q as following.

$$ER(S, s') = wt(s')$$

Then the heuristic algorithm for VSG problem is proposed in algorithm 5

Algorithm 4 Heuristic and Greedy (HG) Algorithm for VSG

Input: $\Gamma = \{G_1, G_2, ..., G_n\}$; Memory Buffer B;
Output: An valid operation sequence S.

```
Create a empty sequence S and a ready queue Q;
for each vertex v in V_Γ do
  if v.outdegree = 0 then
    Add vertex v to Q;
  end if
end for
while Q is not empty do
  m = max{DR(S, s')|s' ∈ Q)}; {Check whether exists direct reduction.}
  if m > 0 then
    let s_{j+1} = s', where DR(S, s') = m;
  else
    m = max{IR(S, s')|s' ∈ Q)}; {Check whether exists indirect reduction.}
    if m > 0 then
      let s_{j+1} = s', where IR(S, s') = m;
    else
      m = max{ER(S, s')|s' ∈ Q)}; {Estimate reduction based on weight.}
      let s_{j+1} = s', where ER(S, s') = m;
    end if
  end if
  Delete s_{j+1} from Q;
  UPDATE(s_{j+1}, S, Γ);
  {Update the sequence S and the ready queue Q.}
end while
```

5. SOLUTION FOR THE JP PROBLEM

Similar to the VSG problem, here we propose a heuristic algorithm to solve the JP problem. Before move on to the detailed algorithm design, we will first start with some preliminaries.

- For any subset $\Gamma_i = \{G_1^{(i)}, G_2^{(i)}, ..., G_{n_i}^{(i)}\}$, we define the file set F_i that includes the files needed to accomplish all jobs in Γ_i. Meanwhile we define file set F_{G_k} for each job G_k in Γ, which includes all the files needed to accomplish G_k.

- For each file set, we define a function *size()* to calculate the cumulative sum of all files in the set. For example, $size(F_{G_k})$ will return the sum of the sizes of all files needed to accomplish G_k.

Now we can discuss how the heuristic algorithm works.

1. Given the job set Γ, first sort all the G_k based the value of $size(F_{G_k})$, where G_1 is supposed to have the largest cost. The new Γ achieved will satisfy that for any G_i and G_j $(i < j)$, $size(F_{G_i}) \geq size(F_{G_j})$.

2. Then the algorithm will start at G_1 and assign all DAGs in Γ into a total m subsets. We define the estimated workload ω to help the decision making.

$$\omega = Cost(\Gamma_i, M_i) - \alpha * size(F_{G_k} \cap F_i)$$

The $Cost(\Gamma_i, M_i)$ indicates the current workload that the server M_i has. $size(F_{G_k} \cap F_i)$ denotes the estimation of the amount of cost can be reduced if assigning G_k to M_i. For each G_k, the algorithm will pick a subset Γ_i having the smallest estimated workload ω and add G_k to Γ_i. The factor α is used to control the

partition strategy. For a larger α, the algorithm will favors the cost reduction can be created by assigning G_k to a server M_i. On the other hand, when α is set to be small, the algorithm will prefer the well-balanced workloads for all the servers.

3. Repeat step 2 until all DAGs in Γ have been partitioned into subsets.

Algorithm 5 Heuristic Algorithm for JP Problem (HP)

Input: Job set $\Gamma = \{G_1, G_2, ..., G_n\}$; server Set $\Omega = \{M_1, M_2, ..., M_m\}$;
Output: m subsets $\Gamma_1, \Gamma_2, ..., \Gamma_m$;

```
Sort all the G_k in Γ with descending size(F_{G_k}) value;
for k=1 to n do
  for i=1 to m do
    Calculate weight w_i= Cost(Γ_i, M_i) − α ∗ size(F_{G_k} ∩ F_i);
  end for
  Assign G_k to the machine with the minimum weight;
end for
```

The key concept here is similar to Graham's List Scheudling algorithm and the Longest Job First (LJF) algorithm, which always schedules the job having longest completion time to the machine with least workload each time. Here, the algorithm considers both workloads and the cost reduction that can be achieved.

6. SIMULATION AND RESULTS

6.1 Simulation Setup

To evaluate the performance of the heuristic algorithm proposed, a set of simulation experiments are conducted. All the simulations are implemented with C++11 standard and conducted under Mac OS environment. Meanwhile, the simulations adopt Boost Graph Library (BGL) to provide the generic interfaces of graph structure.

To evaluate the performance of the HG algorithm, we propose other two algorithms that can generate a valid sequence with the set Γ given. Both of them are based on the topological sorting of a single DAG.

1. For each DAG G_i in Γ, topological sorting will generate a valid sequence S_i. Catenate all the S_i arbitrarily into one sequence S, which will be a valid sequence for the set Γ. We use CA (catenate arbitrarily) as the abbreviation of the solution.

2. Similarly, for each DAG G_i in Γ, apply topological sorting upon G_i to construct the sequence S_i. Different from CA, we apply a round-robin principle to construct the overall sequence S. Figure 6.1 shows how the sequence S is generated. For each round, exact one subjob will be picked from each G_i and put into S until all the subjobs are included. We use RR (round-robin) as the abbreviation of the second solution.

Meanwhile, the upper bound and the lower bound of the cost will be applied as references to the performance evaluation as well. For any sequence S generated, we

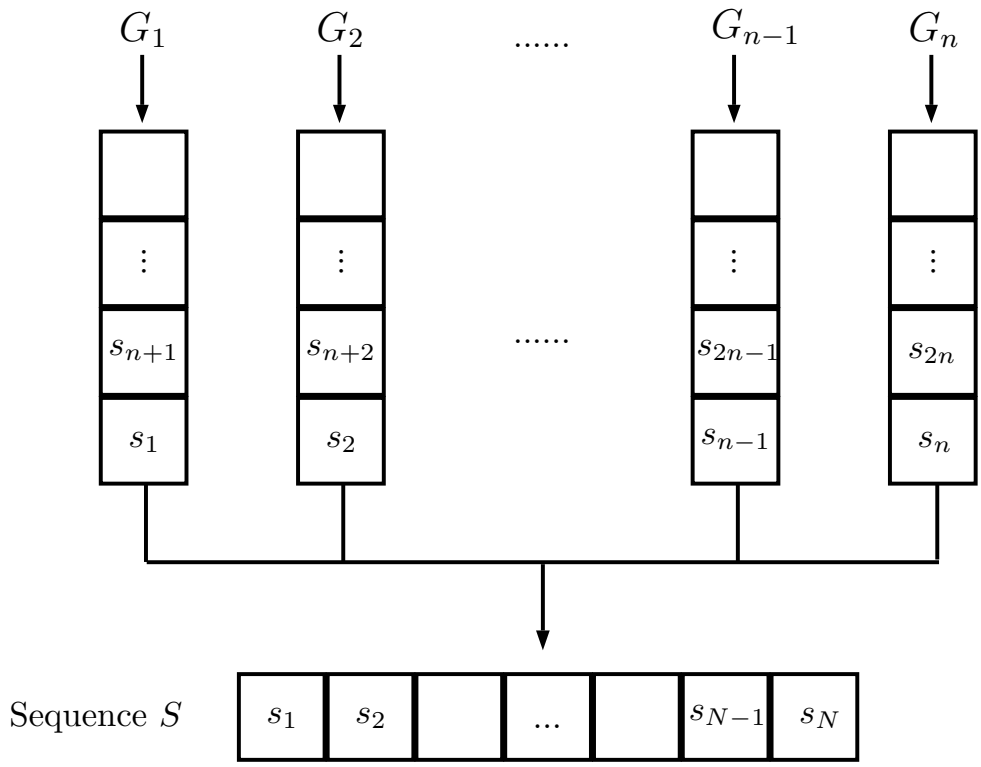

Figure 6.1: Sequence Generation via Round-Robin Scheduling

evaluate the cost of S in the form of

$$P(S) = \frac{Cost_S}{cost\ lower\ bound}$$

The cost lower bound of a sequence S is derived as $\sum_{f \in F_{1,N}} f.size$ in Theorem 4.2, where $F_{1,N}$ is the file set that includes all the target files needed in S, i.e. $F_{1,N} = \bigcup_{i=1}^{N} \{f(s_i)\}$.

Since this research is motivated by the practical scheduling issue in Transparent Computing (TC) environment. The data sets used in the simulation will also consider the real needs in TC. Remind that clients with various types of terminal devices will request for operating systems, drivers, compliers or applications from TC servers. We use the model proposed in Herraiz, et al.[8] to construct the data sets used in

the simulations. Herraiz, et al.[8] presents a complete analysis upon the distribution of file sizes in a linux distribution. The work concludes that the source files in a Linux system follow a lognormal distribution. Therefore we generate the test cases with the size of target files following a lognormal distribution with two distribution parameters μ and σ, where μ represents the mean of the variables, which, in our case, is the average size of the target files, and σ is the standard deviation of the variable's natural logarithm, which will be interpreted as the amount of variation of the sizes of target files used in the simulations.

In sum, we will evaluate the performance of candidate algorithms upon the following parameters.

- The proportion p of all subjobs eligible for cost reduction in set Γ.
- The mean μ of the lognormal distribution of all target files requested by clients.
- The standard deviation σ in target file size distribution.
- The size of memory buffer B.
- The size of DAG set Γ.
- The ratio r that indicates the number of edges connected to each vertex. If $r = 0.3$, it means each vertex are connected to 30% of the total number of vertices in G_i, either as a source vertex or a target vertex.

6.2 Simulation Results and Analysis

6.2.1 The VSG Problem

To begin with, we evaluate the performance of heuristic algorithms with varying average target file size. Seven data sets are created with target file size following a lognormal distribution, where the mean value μ varies from 1 to 4 and the standard

devision is fixed as $\sigma = 0.25$. It indicates that the average target file size will vary between 2.7 MB to 54.6 MB (All the size mentioned in the section will be in the unit of MB). Morever, all cases are simulated with a Γ set with 20 DAGs. Each DAG is composed with 50 vertices and about 375 edges ($r = 0.3$). The buffer size is set to be 1000 and a maximal 30% of total subjobs in Γ can be reduced in cost. The simulation results are shown in Figure 6.2 and more detailed simulation results are included in the Table 8.1 in the Appendix section.

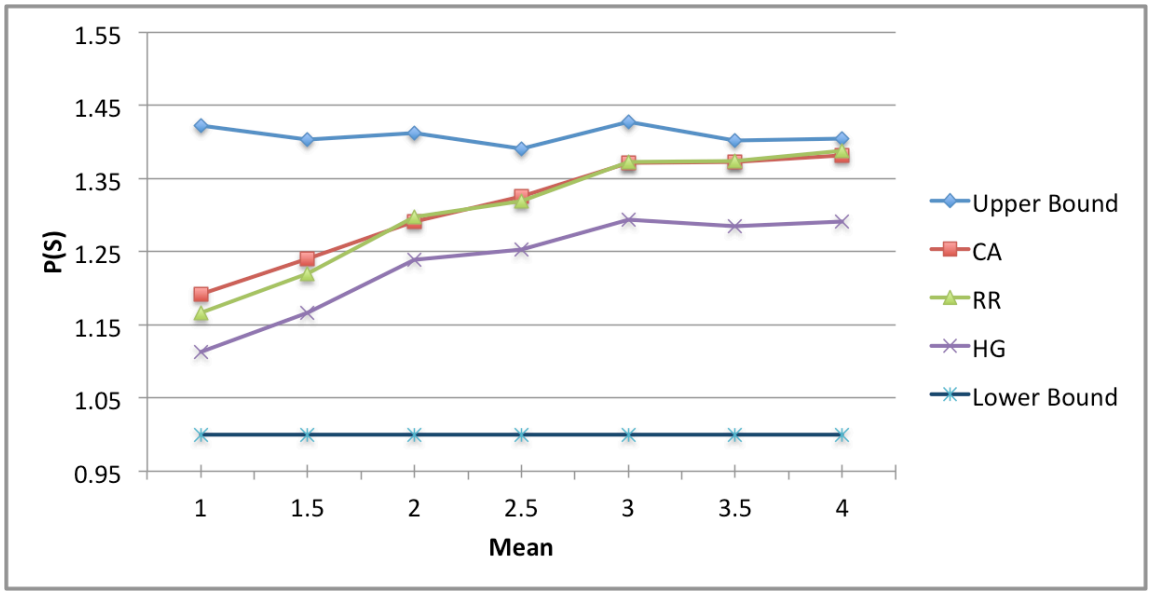

Figure 6.2: Performance of Candidate Solutions upon Mean μ.
$\sigma = 0.25$, $|\Gamma| = 20$, $|V_{G_i}| = 50$, $r = 0.3$, Buffer Size $= 1000$, $p = 30\%$.

As shown in Figure 6.2, the sequences generated by *HG* have costs that are 10% higher than lower bound when average size is small. And it is about 30% higher when the mean value increases to 4. When the average target file size is increasing, the number of files can be hold in the memory buffer will be decreased. Therefore, all the heuristic solutions have decreasing performance with the increasing μ. Compared to

CA and *RR*, the performance of *HG* is better by $7\% - 10\%$ in reference to the lower bound. Larger the mean value is, better performance improvement can be achieved by *HG* compared to *CA* and *RR*.

Furthermore, consider a fixed mean value $\mu = 3$, we will explore the impact of the standard deviation. Six test cases are conducted with the standard deviation that varies from 0 to 1.25. All other parameters are kept as same as the previous simulation.

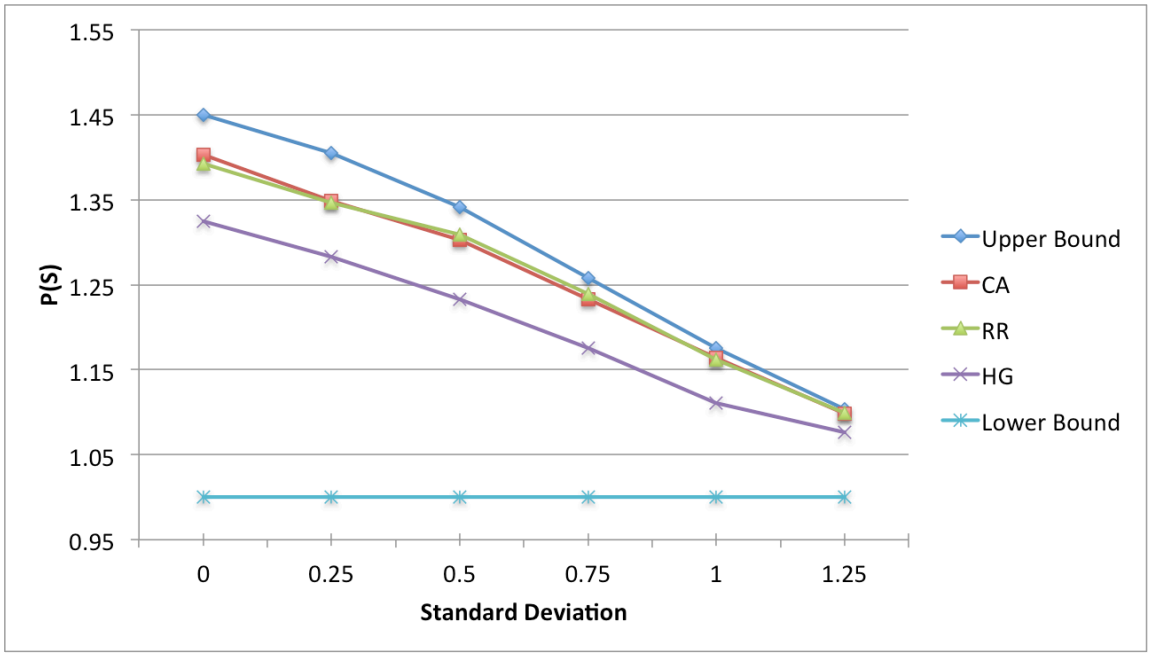

Figure 6.3: Performance of Candidate Solutions upon Standard Deviation σ. $\mu = 3$, $|\Gamma| = 20$, $|V_{G_i}| = 50$, $r = 0.3$, Buffer Size $= 1000$, $p = 30\%$.

As shown in the Figure 6.3, when the standard deviation increases, the performance of all three algorithms will converge to the upper bound. Compared to *CA* and *RR*, *HG* will have more obvious advantage when the standard deviation is low.

Besides the distribution of the target file size, the features of the DAG set Γ will

also affect the performance of algorithm. Starting with the size of Γ, six simulation tests are conducted with the size of Γ scale from 10 to 60. The size of target files follows a lognormal distribution with $\mu = 3$ and $\sigma = 0.25$. Other parameters are kept as same as previous cases.

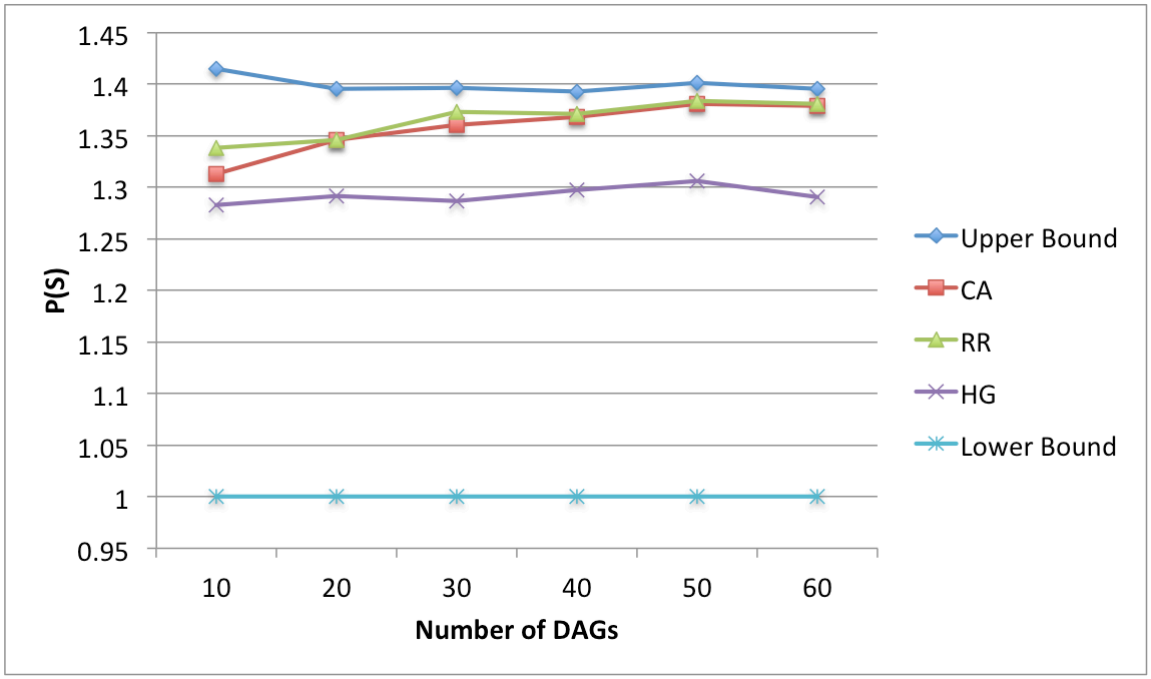

Figure 6.4: Performance of Candidate Solutions upon the Size of Γ.
$\mu = 3$, $\sigma = 0.25$, $|V_{G_i}| = 50$, $r = 0.3$, Buffer Size $= 1000$, $p = 30\%$.

From Figure 6.4, we can find that the performance of *HG* is not significantly by the size of Γ. While for *CA* and *RR*, their performances will converge to the upper bound when the set Γ increases in size.

In addition, within a single DAG G_i, the number of vertices and edges will also affect the performance, since the dependency relations are extremely essential to the generation of a valid sequence. Figure 6.5 includes six cases, where the number of vertices in each DAG G_i varies from 10 to 110 and the ratio r is kept as constant.

All three algorithms will be affected by the increasing vertices number. When the DAG has fairly small number of vertices, *HG*'s performance is fairly close to the lower bound.

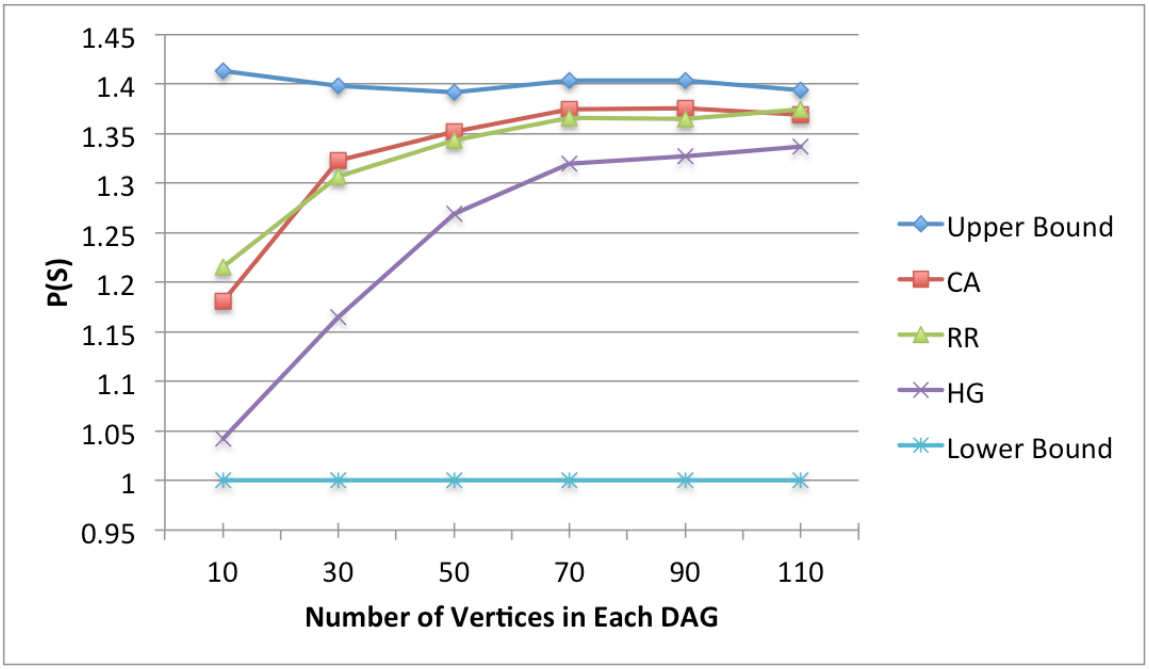

Figure 6.5: Performance of Candidate Solutions upon the Size of Each DAG G_i. $\mu = 3$, $\sigma = 0.25$, $|\Gamma| = 20$, $r = 0.3$, Buffer Size $= 1000$, $p = 30\%$.

The ratio r is another important parameter concerning the structure of a *DAG*. Note that the ratio r indicates how many edges that a single vertex is connected with, either as a source vertex or a target vertex. Each vertex is connected via a total $r * |V_{G_i}|$ edges, where $|V_{G_i}|$ is the number for vertices in G_i. Figure 6.6 presents a full survey with the ratio r varying from 0.1 to 1.0. It can be concluded that compared to *CA* and *RR*, *HG* will have more advantage upon DAGs with simpler dependency relations. Both *CA* and *RR* are not edge-sensitive as indicated in the Figure 6.6.

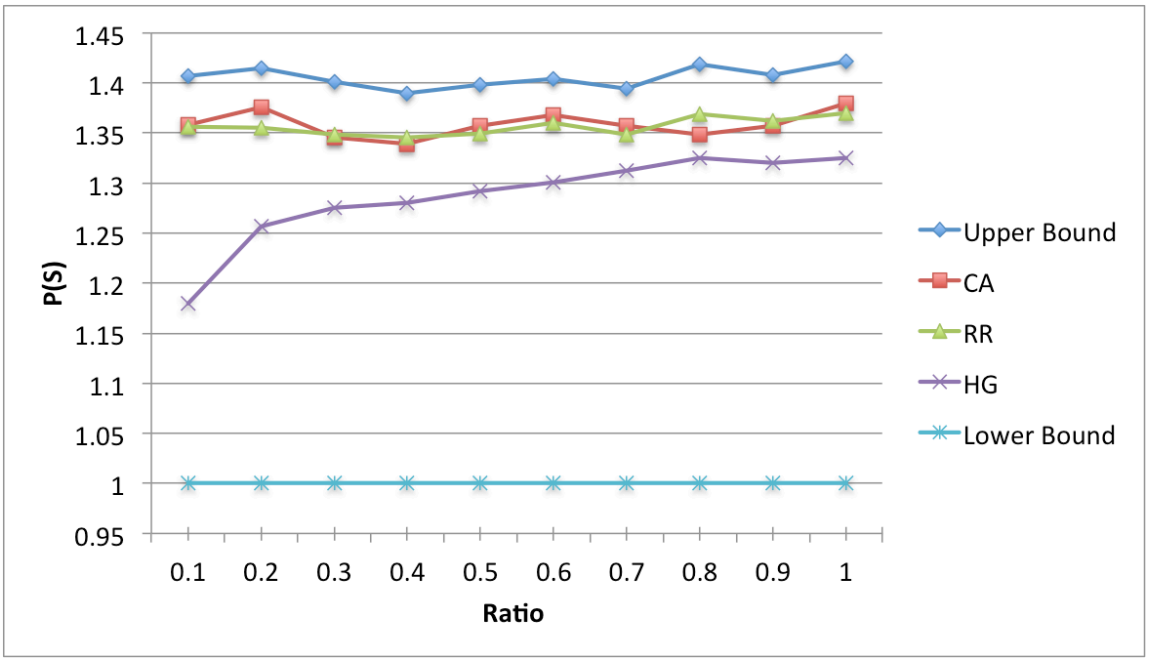

Figure 6.6: Performance of Candidate Solutions upon the Amount of Edges Connected to Each Vertex (r).
$\mu = 3$, $\sigma = 0.25$, $|\Gamma| = 20$, $|V_{G_i}| = 50$, Buffer Size $= 1000$, $p = 30\%$.

The next parameter discussed will be the buffer size. Figure 6.7 covers cases with buffer size ranging from 200 to 2000. HG has a average $\Delta P(S) = 0.07$ better than CA and RR and its performance converges to the lower bound as the buffer size increases.

Lastly, we analyze the performance of candidate solutions upon the proportion p, which indicates the number of subjobs that are eligible for cost reduction. More generally speaking, the proportion p can be interpreted with upper bound and lower bound as

$$p = 1 - \frac{lower\ bound}{upper\ bound}$$

Figure 6.8 presents simulation cases with proportion p ranging from 0.2 to 0.8. The

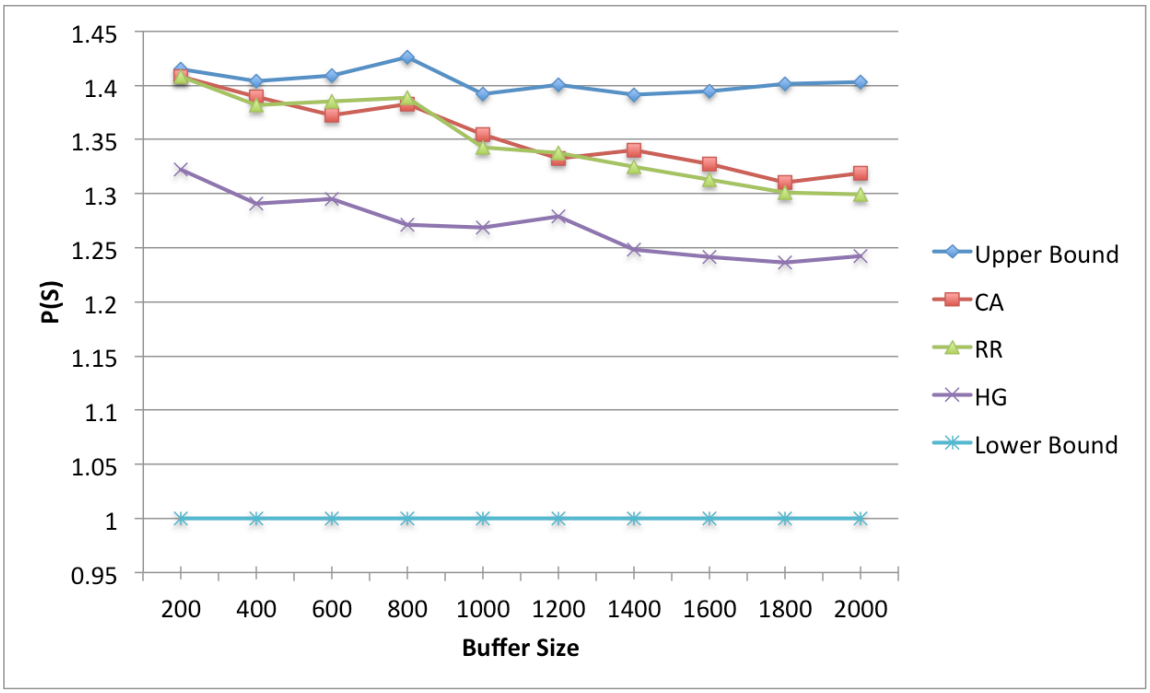

Figure 6.7: Performance of Candidate Solutions upon the size of Memory Buffer. $\mu = 3$, $\sigma = 0.25$, $|\Gamma| = 20$, $|V_{G_i}| = 50$, $r = 0.3$, $p = 30\%$.

performance of *HG* indicates its potential in cost reduction with a high redundancy in target files.

In summary, according to all the simulation results above, we can derive the following conclusions concerning three candidate solutions. For *HG*, it shows great performance in the following cases.

- Compared to *CA* and *RR*, *HG* brings more reduction in cost in most cases. Especially when the redundancy in target files is frequent, *HG* will take more benefits from it.

- *HG* can generate sequences with costs close to the lower bound for those DAGs with less dependencies.

- *HG* has a stable performance independent of the size of Γ.

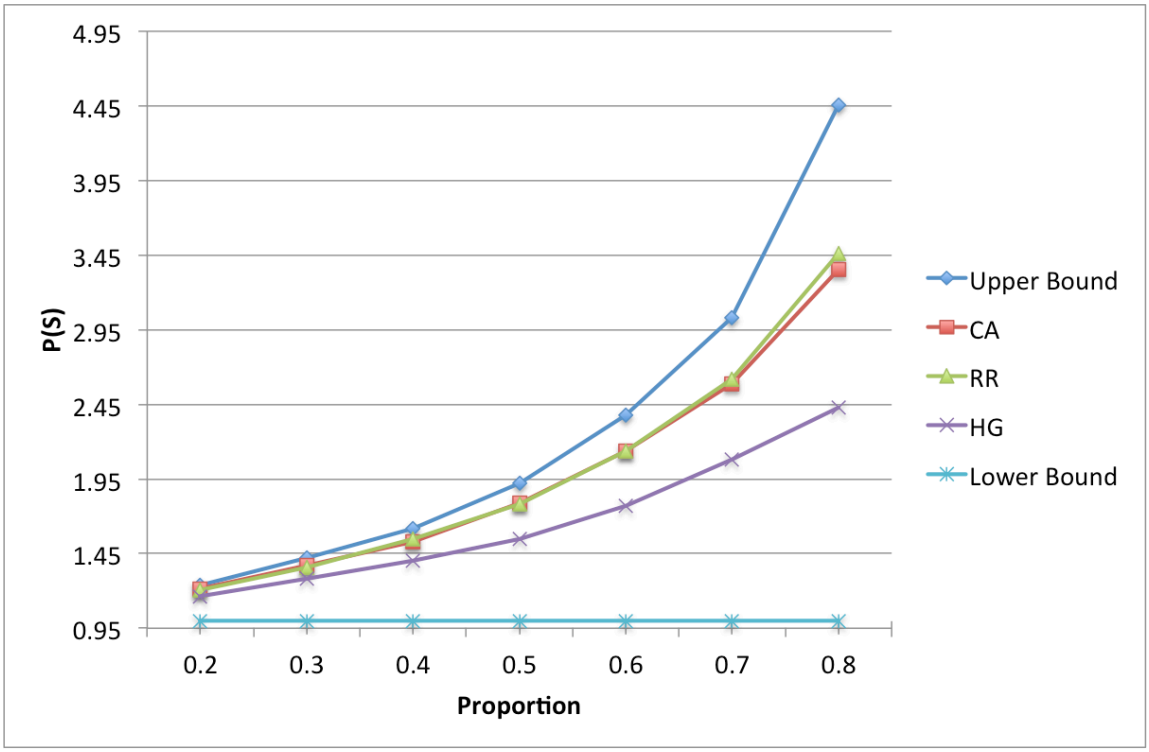

Figure 6.8: Performance of Candidate Solutions upon proportion p.
$\mu = 3$, $\sigma = 0.25$, $|\Gamma| = 20$, $|V_{G_i}| = 50$, $r = 0.3$, Buffer Size = 1000.

- *HG* has more advantages over *CA* and *RR* when the average file size is large and the standard deviation is small.

Moreover, concerning *CA* and *RR*, we can derive the following conclusion.

- *CA* and *RR* are not edge-sensive. They will have a more stable performance compared to *HG* in case the *indegree* and *outdegree* for vertices are diverse.

- *CA* and *RR* share almost same performance and complexity in sequence generation. Their sequence generation process is faster than the one of *HG*. Especially when the set Γ includes more DAGs and each DAG has more vertices.

6.2.2 The JP Problem

We perform a set of simulations upon the heuristic algorithm proposed for the JP problem. The setup is similar to the simulations of VSG problem. We use Longest Job First (LJF) algorithm as a reference when we evaluate the heuristic (HP) algorithm proposed in this work. To has better focus on the partition problem, we define the following settings for sequence generation problem identically for each underlying servers.

- Each server will use HG algorithm proposed in this work to calculate the actual cost for each Γ_i.
- The buffer size will be set to be 1000 MB.
- The target file size follows a lognormal distribution with mean $\mu = 3$ (about 20MB) and standard deviation $\sigma = 0.25$.

Then the first parameter this section will cover is the parameter α. We want to figure out how the value of α will affect the final cost for file fetching. Here we assume we have 50 DAGs need to be distributed and each DAG having 40 vertices. The redundant portion of identical target files p is set to be 0.3, where indicates half of the subjobs may be reducible in cost. We test the partition algorithm upon 5 server, where make a Job-to-Server ratio $n/m = 10$.

Figure 6.9 includes the cost for the M_i, which has the largest cost after the partition. The two algorithms have really close performance when α is small and *HP* algorithm does have an advantage compared to *LJF*. However, when the α gets larger, the *HP* starts to lose the good workload balance between underlying servers.

However, according to the results in Figure 6.10, *HP* have a better average cost compared to *LJF* in most cases. As the α increases, the *HP* will have a even better

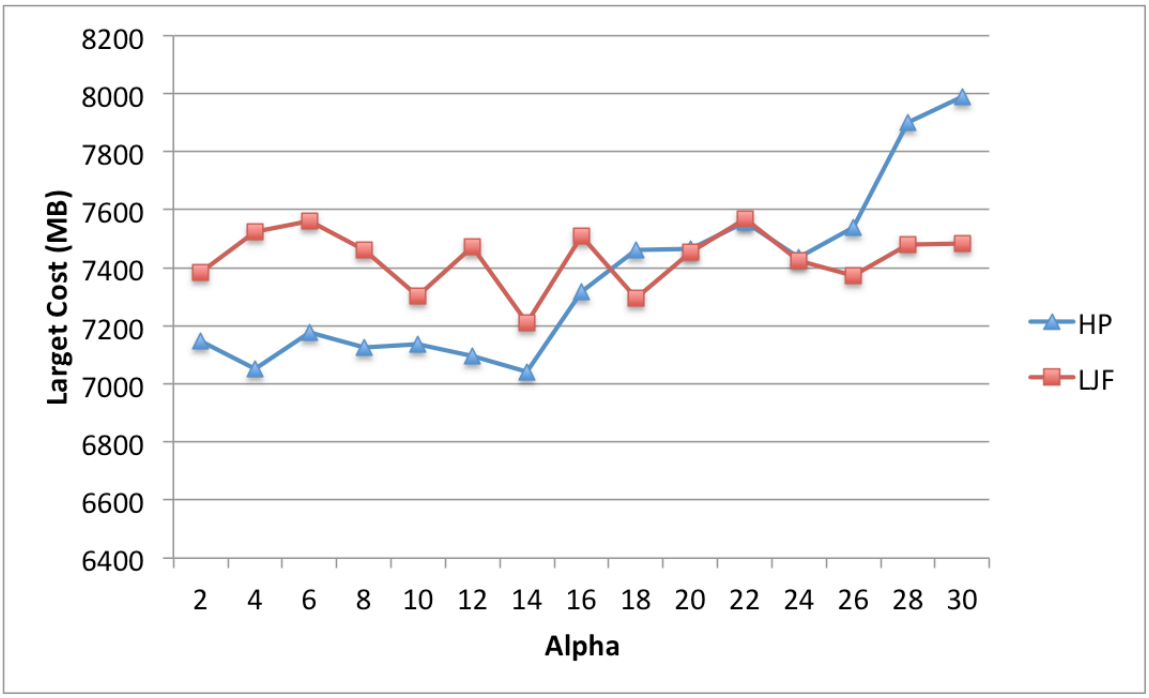

Figure 6.9: The Largest Cost for a Single Server on α

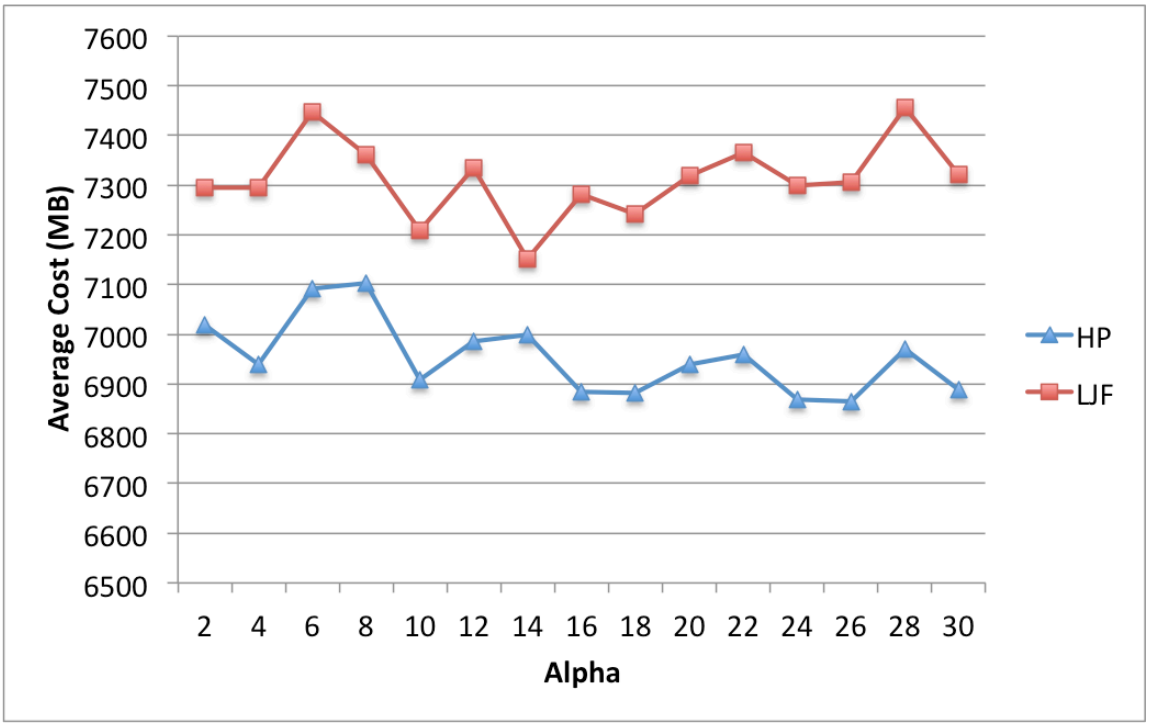

Figure 6.10: The Average Cost for All Servers on α

cost reduction referring to *LJF*.

Another important parameter we think may relate to the performance will be the Job-to-Server ratio n/m. We take a α value equal to 10 and keep other parameters as the same as the previous simulation.

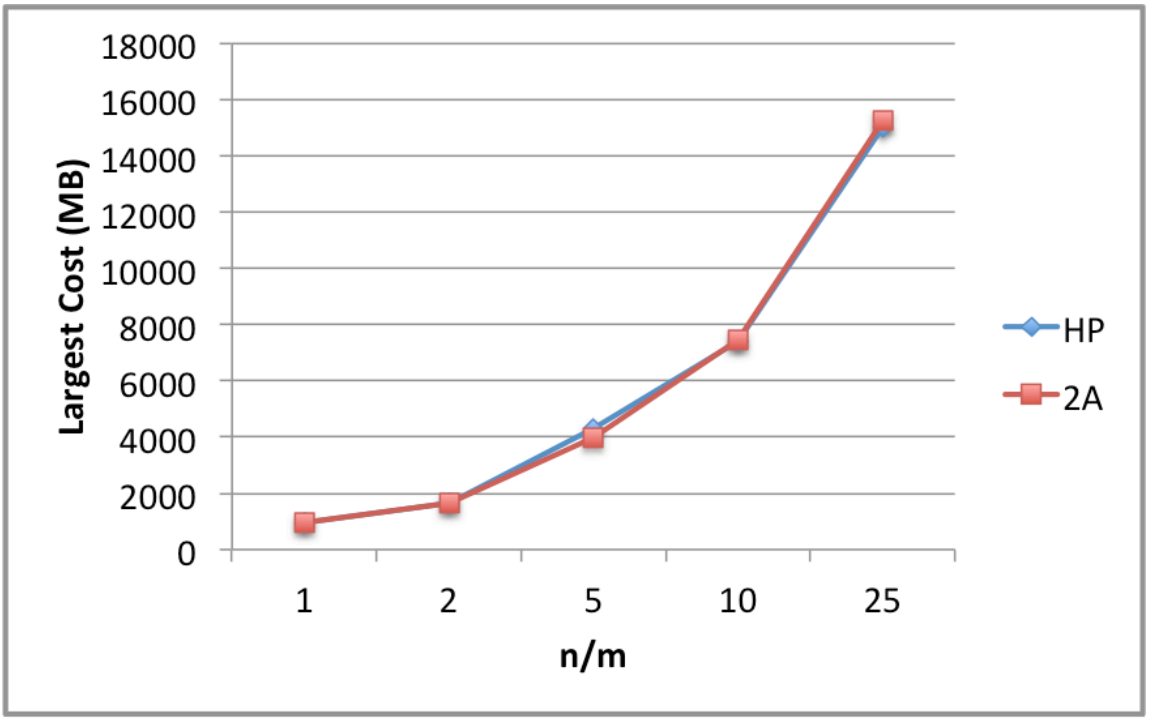

Figure 6.11: The Largest Cost for a Single Server on n/m

Figure 6.11 and 6.12 both show the relation between the performance of the candidate algorithms and the ratio n/m. While both algorithms have a close performance in the largest cost for a single server, *HP* does have better cost reduction when each machine is responsible for more jobs.

In summary, the heuristic algorithm *HP* proposed for the job partition algorithm have a close performance to the Longest Job First algorithm concerning the workload balance. Beyond that, *HP* does reduce the amount of disk IO cost via assigning each

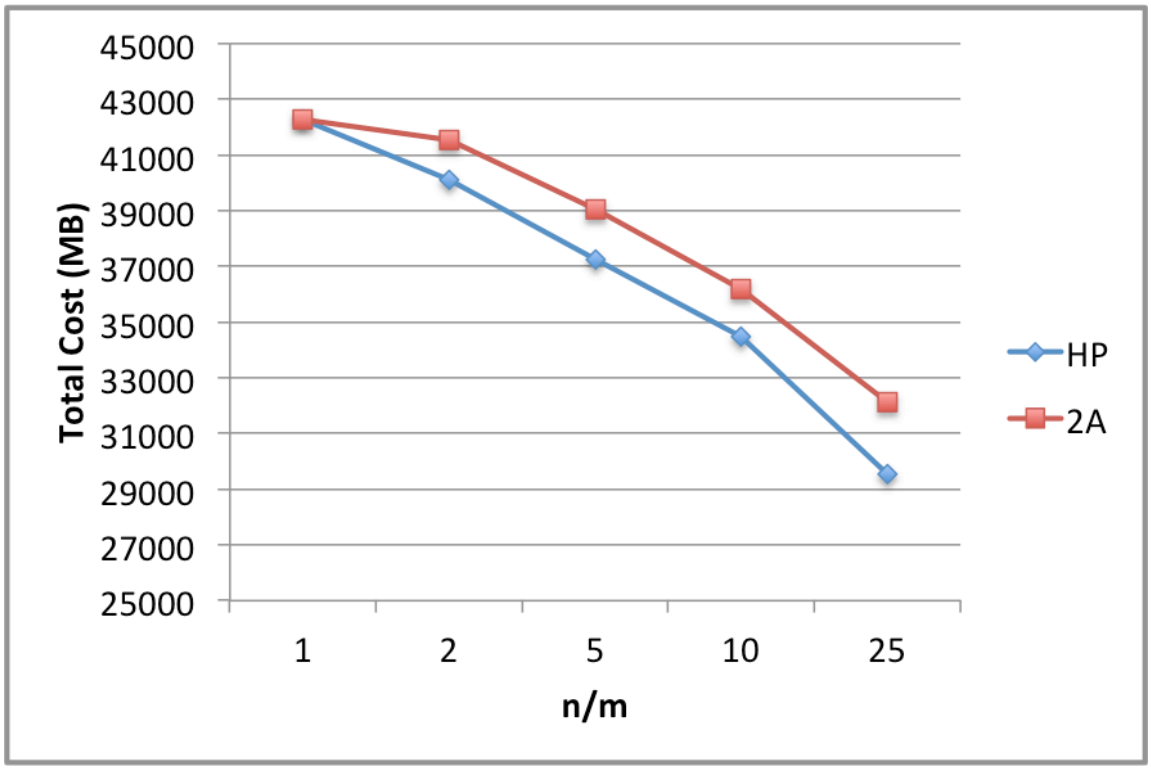

Figure 6.12: The Average Cost for All Servers on n/m

job to a server, where exits extra disk IO cost reduction for the file fetching fetching process.

7. FUTURE WORK

7.1 More Topics for File Fetching Process

This works aims at proposing the model where the disk IO cost can be reduced via optimized scheduling. We use heuristic algorithms to validate and show the value of the problems we defined. However, these heuristic algorithms have great room to be further optimized concerning the running time complexity. The followings are some interesting topics that are valuable upon this work.

- For the single-server VSG problem, each DAG can be scheduled into a valid sequence S_i and the overall sequence S for the set Γ can be considered as the result of merging all the S_i together. To minimize the disk IO cost of the sequence S, the subjobs requesting for identical target files should be scheduled to be next or close to each other in S. Therefore, the sequence generation can be interpreted as a sequence alignment problem with the valid sequences for each DAG are given.

- Beyond the sequence alignment mentioned above, the permutation of difference valid sequences from difference DAGs is also an interesting topic to cover. Upon having a valid sequence S_i generated from the DAG G_i, some subjobs in S_i can be swapped while the sequence S_i' after the swapping is still valid. Therefore, exploring the permutations of sequences S_i for each DAG to achieve a minimized disk IO cost in the sequence alignment will be a valuable topic.

- For the multi-server scheduling, we also found interpreting this with a limited-budget in disk IO can bring valuable perspectives. More than a simple makespan problem, now we can define a budget for each DAG and each underlying server

has a price reflecting its current workload. Considering the disk IO reduction as the reward, the target of the algorithm will be maximizing the reward via assigning each DAG to servers, where the budget of the DAG can afford the price of the server it will be assigned to.

7.2 Weight Assignment

For the *HG* algorithm proposed, the *estimation reduction* is derived from the weight of each subjob vertex. The core idea of the weight of s_i is to indicate how many subjobs are depending on s_i. In section 4.2, we define the weight of each subjob s_i is equal to the *indegree* of $\pi(s_i)$. There actually exist alternative methods in defining the weight here. All of them are stick to the core idea we have here and can fit into the *HG* algorithm proposed.

1. Let $wt(s_i)$ equal to the cumulative sum of the target file size of all the subjobs directly depend on s_i.

2. Let $wt(s_i)$ to the number of subjobs s_j, which has a path starting from s_j and ended at s_i. In other words, $wt(s_i)$ can be equal to the total number of subjobs directly or indirectly depend on s_i.

3. Combine 1 and 2 together, $wt(s_i)$ can be defined as the cumulative sizes of all the subjobs directly or indirectly depend on s_i.

It need to be noticed that some methods of weight assignment may have favors in specific types of subjobs. It will be an interesting topic to explore the features of different methods.

7.3 Beyond File Fetching

Since this work is motivated by the real need in Transparent Computing, the file fetching process is the major focus. However, the core idea of this work can

be further extended beyond file fetching process. The cloud will be faced with all kinds of job requests and the overlap between these jobs can be significant. When scheduling these jobs, taking benefits for these overlapped jobs can a breakthrough point to the performance optimization of the cloud.

Meanwhile, the model proposed in this work can be further extended to consider not only just the reading/fetching of files, but also updating/writing files in the cloud. The dependency will become more complicated and there will be more mutex requirements need to be satisfied.

8. CONCLUSION

The whole discussion starts from a core idea — dependent data fetching can be optimized via benefiting from identical data requests. This idea comes from the recent studies on transparent computing, where clients continuously request large number of files from the cloud and some commonly used files (like OS and software source files) may be requested by several clients concurrently and repeatedly. To validate the value of this idea, we define the valid sequence generation problem, which aims at reducing the disk IO cost of file fetching under the model abstracted from the transparent computing architecture. A heuristic algorithm is proposed specifically for the problem and the simulation results indicate the optimization in disk IO cost brought by the algorithm proposed. Beyond that, a Job Partition Problem is defined for the multi-server scheduling and its validity and correctness is shown with the heuristic algorithm specifically proposed for the problem. Further research can be conducted following the core idea here and bring more efficient solutions for file fetching process in cloud environment.

REFERENCES

[1] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 285–300. USENIX Association, 2014.

[2] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[3] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[4] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[5] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *American mathematical monthly*, pages 9–15, 1962.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *ACM SIGOPS operating systems review.*, 37(5), 2003.

[7] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the*

3rd USENIX conference on Hot topics in cloud computing, pages 3–3. USENIX Association, 2011.

[8] Israel Herraiz, Daniel M Germán, and Ahmed E Hassan. On the distribution of source code file sizes. In *ICSOFT (2)*, pages 5–14, 2011.

[9] Robert W Irving. An efficient algorithm for the stable roommates problem. *Journal of Algorithms*, 6(4):577–595, 1985.

[10] Selmer Martin Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, 1(1):61–68, 1954.

[11] Ren Ju, Zhang Yaoxue, and Chen Jianer. Analysis on the scheduling problem in transparent computing. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1832–1837. IEEE, 2013.

[12] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[13] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20. ACM, 2010.

[14] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[15] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. Ieee, 2009.

[16] Eric Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European journal of Operational research*, 47(1):65–74, 1990.

[17] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[18] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM transactions on computer systems (TOCS)*, 21(2):164–206, 2003.

[19] Hong Xu and Baochun Li. Egalitarian stable matching for vm migration in cloud computing. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 631–636. IEEE, 2011.

[20] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[21] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[22] Yaoxue Zhang and Yuezhi Zhou. Transparent computing: a new paradigm for pervasive computing. *Ubiquitous Intelligence and Computing*, 1-11, 2006.

[23] Yuezhi Zhou and Yaoxue Zhang. *Transparent computing: concepts, architecture, and implementation*. Cengage Learning Asia, 2010.

[24] Yuezhi Zhou, Yaoxue Zhang, Hao Liu, Naixue Xiong, and Athanasios V Vasilakos. A bare-metal and asymmetric partitioning approach to client virtualization. *Services Computing, IEEE Transactions on*, 7(1):40–53, 2014.

APPENDIX

All the detailed simulation results used in section 5 will be included here. All data will be presented

Table 8.1: $P(S)$ Values for Figure 6.2

Mean	Upper Bound	MS	RR	HG
$\mu = 1.0$	1.422553752	1.192189557	1.166739798	1.113207547
$\mu = 1.5$	1.403819918	1.240381992	1.219918145	1.166166439
$\mu = 2.0$	1.412428398	1.29178962	1.297517792	1.239715327
$\mu = 2.5$	1.390916808	1.325870119	1.319927844	1.25360781
$\mu = 3.0$	1.427411168	1.372182741	1.373401015	1.293265651
$\mu = 3.5$	1.402521746	1.372955071	1.374551113	1.28433485
$\mu = 4.0$	1.404404208	1.381967213	1.388182041	1.29168094

Table 8.2: $P(S)$ Values for Figure 6.3

Standard Deviation	Upper Bound	MS	RR	HG
$\sigma = 0$	1.449275362	1.402898551	1.392753623	1.324637681
$\sigma = 0.25$	1.4050895	1.348650815	1.346112744	1.282594176
$\sigma = 0.5$	1.341178552	1.303014216	1.308499971	1.233233056
$\sigma = 0.75$	1.25832497	1.233053554	1.239211751	1.175348965
$\sigma = 1$	1.175679655	1.164360258	1.162299212	1.110642196
$\sigma = 1.25$	1.10354928	1.098423305	1.099064052	1.076020046

Table 8.3: $P(S)$ Values for Figure 6.4

Γ Size	Upper Bound	MS	RR	HG
10	1.414852411	1.313075998	1.338319754	1.282890343
20	1.395581659	1.346459321	1.346058867	1.29179737
30	1.396700396	1.360844699	1.373119226	1.28658161
40	1.392916599	1.36833687	1.37078505	1.29714379
50	1.40118997	1.38079048	1.384296643	1.306337654
60	1.395954718	1.379051375	1.381133831	1.290369747

Table 8.4: $P(S)$ Values for Figure 6.5

DAG Vertex Size	Upper Bound	MS	RR	HG
10	1.412874958	1.1809909	1.2157061	1.041793057
30	1.398634512	1.323422531	1.307234886	1.165180046
50	1.391372243	1.351616891	1.34341857	1.269308348
70	1.403874321	1.375147649	1.366406804	1.319253485
90	1.403318745	1.375747629	1.364697301	1.327534646
110	1.394437421	1.36966468	1.374119559	1.336764794

Table 8.5: $P(S)$ Values for Figure 6.6

Ratio r	Upper Bound	MS	RR	HG
0.1	1.406620324	1.358556769	1.356636875	1.179410791
0.2	1.414622642	1.375471698	1.355727763	1.25680593
0.3	1.401617782	1.345126924	1.34880968	1.275088781
0.4	1.389775317	1.339238033	1.345555194	1.279908824
0.5	1.398130597	1.356886071	1.349434604	1.291718413
0.6	1.404527754	1.367896508	1.36030625	1.300706224
0.7	1.394312169	1.35734127	1.348148148	1.3125
0.8	1.418434479	1.348885607	1.36929377	1.325523631
0.9	1.408239451	1.357442837	1.361842544	1.32064529
1	1.421320477	1.379603589	1.370429892	1.324829249

Table 8.6: $P(S)$ Values for Figure 6.7

Buffer Size	Upper Bound	MS	RR	HG
200	1.415104482	1.408654169	1.408385406	1.322045287
400	1.403615655	1.389444408	1.381895239	1.290841666
600	1.408838671	1.372735174	1.384970248	1.295045798
800	1.425884582	1.382382789	1.388606995	1.270888303
1000	1.391745298	1.354900828	1.342769112	1.268695038
1200	1.400961412	1.332674832	1.337284341	1.27887528
1400	1.391459537	1.340061003	1.324615484	1.248231553
1600	1.394804005	1.327138276	1.313068516	1.241410902
1800	1.401174685	1.31030159	1.300666535	1.235992873
2000	1.403176502	1.318979266	1.29917597	1.242025518

Table 8.7: $P(S)$ Values for Figure 6.8

Proportion r	Upper Bound	MS	RR	HG
0.2	1.234543208	1.215012648	1.204894406	1.164244956
0.3	1.418482293	1.369376054	1.358381113	1.283777403
0.4	1.618259188	1.527040619	1.547466151	1.400077369
0.5	1.920405599	1.791636755	1.78436827	1.54971285
0.6	2.380286455	2.1401827	2.1401827	1.771174016
0.7	3.031047865	2.589047003	2.616357625	2.078913325
0.8	4.456661817	3.356059031	3.463936811	2.425483268

Table 8.8: $P(S)$ Values for Figure 6.9 and 6.10

α	HP(max)	2A(max)	HP(avg)	2A(avg)
2	7148	7385	7018	7294.8
4	7053	7523	6940.4	7296
6	7177	7562	7093	7448.4
8	7126	7461	7102.2	7362.2
10	7138	7303	6909.4	7208.8
12	7097	7473	6986.2	7335.6
14	7042	7209	6998.2	7151.4
16	7318	7509	6883.6	7282.4
18	7461	7294	6883	7241
20	7467	7455	6939.8	7319.4
22	7559	7567	6960	7366.6
24	7437	7424	6868.2	7299.8
26	7538	7373	6865.2	7305.4
28	7899	7480	6971.4	7457
30	7989	7482	6888	7322

Table 8.9: $P(S)$ Values for Figure 6.11 and 6.12

n/m	HP(max)	2A(max)	HP(total)	2A(total)
1	962	962	42263	42263
2	1648	1662	40103	41550
5	4303	3977	37242	39067
10	7397	7470	34492	36190
25	15033	15270	29520	32120